Hacking With Monads

FUNCTIONAL PROGRAMMING FOR THE BLUE TEAM

DEF CON 27

AUGUST 10, 2019

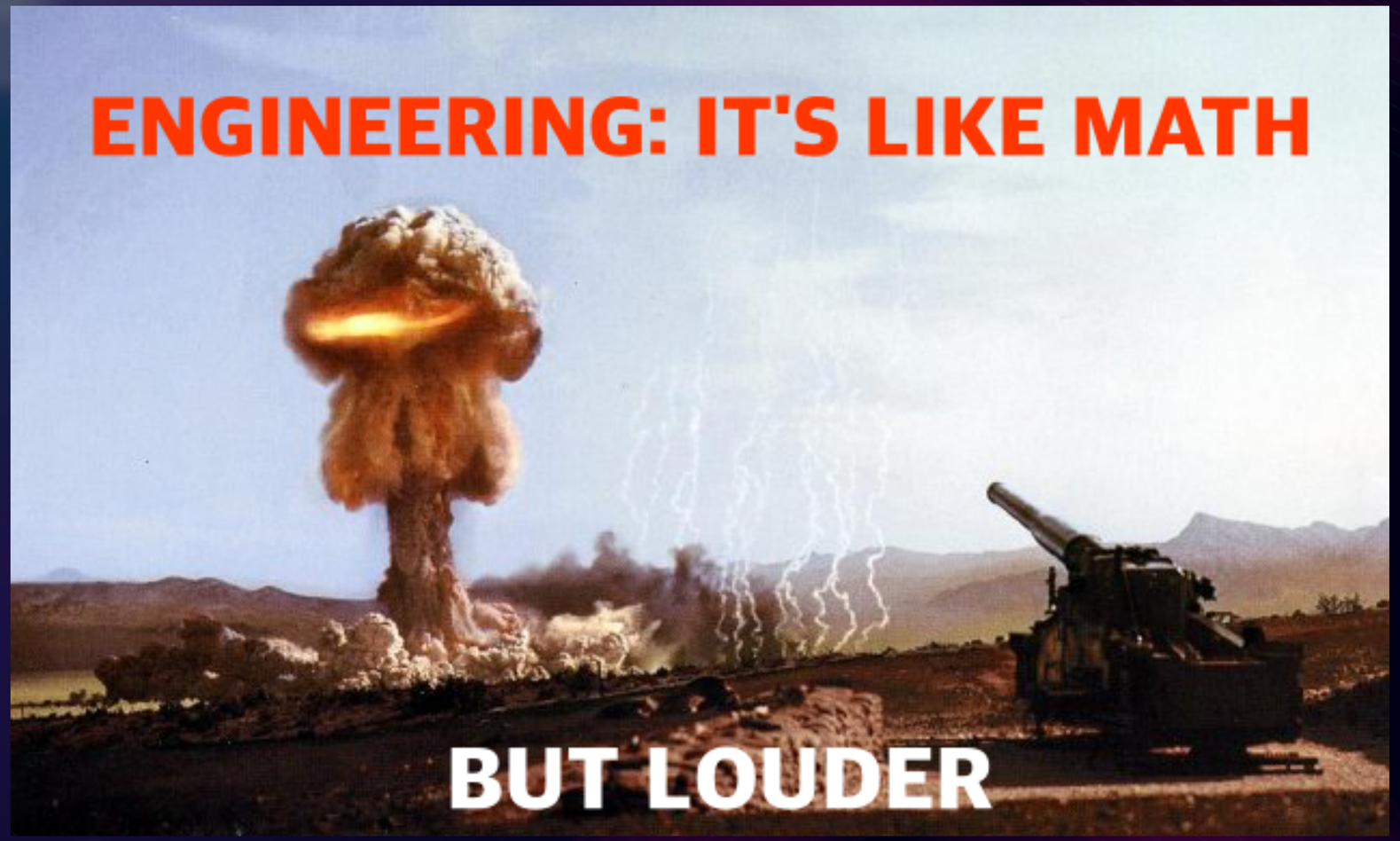
Functional Programming: WTF?

There **are** two tribes of programming philosophy:

- Engineers
- Mathematicians

Engineers

- Built the hardware
- Focus on practical results
- Deal with real-world constraints
  - time
  - materials
  - tools
- Will create memes like this:



Mathematicians

- Created the language
- Focus on theoretical truth
- Deal with ideas and possibilities
  - proofs
  - equations
  - logic
- Will tell jokes like this:

A physicist, a biologist, and a mathematician are sitting in a cafe. Across the street is a house. As they sit talking, two people go into the house. Momentarily, three people come back out.

The physicist says, "our initial count must have been incorrect."
The biologist says, "they must have reproduced."

The mathematician says, "now... if one person goes back in, the house will be empty."

Programming Languages

From The Engineering Tribe:

- Fortran
- ALGOL
- COBOL
- Pascal
- C, C++, Java, et al.

Languages from the Engineering tribe are designed to let you, the programmer, tell the machine exactly what, how and when to do something.

Programming Languages

From The Mathematics Tribe:

- Lisp, Scheme
- ML
- Erlang
- Haskell
- Clojure

Languages from the Mathematics tribe will let you, the programmer, describe a system by defining and combining logical statements:

Functions.

Thinking Mathematically

Mathematicians play by different rules than engineers.

The rest of us can say:

Let a = 10

A mathematician will be fine with the first statement.

and then, later:

But they will not tolerate the second one.

a = a + 10

Variables

To a mathematician, variables are just unknown values. We don't know what they are yet, so we represent them with symbols.

To a software engineer, variables are *locations for storing data*.

They exist in the real world.

Their contents can change.

A Word About Equality

a = 256

For an engineer, the above statement reads like an assignment: store the value 256 in the memory that 'a' is pointing to.

For a mathematician, the ‘=‘ symbol means equivalence: *a* is interchangeable with *256*, anywhere that *a* is found.

A Word About Equality

That interchangeability means that one of those symbols can be substituted for the other, anywhere in your program, and the *output of the program will be the same*.

There's a name for that: *referential transparency*.

It's a good thing.

Programs

Programs don't exist in mathematics.

Symbols exist.

Expressions exist.

Programs are real-world entities. They have behavior that changes based on their internal state.

A program, just like the computer it runs on, is a state machine.

State Machines

A state machine will change behavior based on its rules and input.

Programmers define those rules.

Failing to anticipate any input for any program state can create behavior that is unwanted or undefined.

Program Complexity

The complexity of a program arises from the total number of internal states it can have.

Managing complexity becomes harder as the total number of possible states rises.

This number rises as you add features.

What Have We Tried?

- 1930s: Turing Machine and Lambda Calculus
- 1950s: COBOL and Fortran introduced
- 1960s: Simula, the first object-oriented language
  Edsger Dijkstra writes "GOTO Considered Harmful"
- 1970s: Alan Kay and team, inspired by Simula, create and refine
  Smalltalk at Xerox Palo Alto Research Center

What Have We Tried?

- 1980s: 4th Generation Language builders try to eliminate programmers.
- 1995: *Design Patterns* by Gamma, Helm, Johnson, Vlissides
- 1995-2005: ZOMG, DESIGN PATTERNS
- 2005-06: We can't make the CPU faster without burning down your office, so here's multiple cores. Learn concurrency. Good luck. -- AMD and Intel

What Have We Tried?

- Object-oriented design tries to manage complexity by breaking apart a program into a community of miniature programs, each of which contains its own data and methods for doing its job.

- Objects in a system communicate by passing messages to each other. *This was Alan Kay's main idea,* but people focused on everything else.

- Multiple cores?? I'm supposed to do multithreaded everything now??

What Have We Tried?

- Objects were supposed to be simple, robust and easy to test.
- In reality, managing the complexity of objects turns out to be non-trivial.
- Data that can be changed can also be corrupted.
- Code turns out to be just another kind of data.

Side Effects

A side effect is any change in the state of your program or its environment that happens as a *byproduct* of your code's execution.

It's impossible to eliminate side effects altogether (if we did, our program would never be able to do anything.)

We need side effects. But we also need ways to handle them better.

Reducing Side Effects

Each part of your program that creates or manipulates state will become harder to understand and debug as complexity increases.

Every bit of code that creates or changes program state expands the potential attack surface of your program.

Minimizing the creation and manipulation of internal state is not trivial. But if you can do it, it will yield more predictable, more reliable code.

A Taste of Pure Functions

A function is an expression that describes a unit of logic in our program.

A *pure* function is one that does its job without creating any side effects.

```
factorial 0 = 1
factorial n = n * factorial(n - 1)
```

The above code computes a result from its input, and returns it, without affecting anything else. No matter what happens, it will always behave the same way.

A Word About Types

Over time, it's very common to grow accustomed to the meaning of "data type" as a flavor of data.

Strictly speaking, a type is a *set*.

The type *Integer* is the set of all integers.

The type *String* is the set of all possible combinations of characters, up to the maximum allowed string length.

A Word About Types

When we declare a variable: Var a : Int = 256

We are saying two things:

- a is a member of the set of all integers
- the value of a is 256 (or, a is interchangeable with 256. (Referential transparency, remember?)

Mutability vs Immutability

Functional style encourages you to use immutable items whenever possible, and mutable items only when you have to. In Python, though, this isn't quite so simple. Have a look:

Immutable items in Python:

- int
- float
- bool
- str
- tuple
- frozenset

Mutable items in Python:

- list
- set
- dict

Mutability vs Immutability

Everything in Python is an object.

This can be a rude awakening if you aren't aware of the implications.

If you say: `a = 'abcd'`

and then say: `a = a + 'efgh'`

and finally, `print(a)`

you will see the output `'abcdefgh'`.

Mutability vs Immutability

How did we re-assign a value to an immutable object?

We didn't. We had a reference (a) that pointed to an immutable object ('abcd').

Then, we created a new immutable object (a + 'efgh') and pointed our reference at it.

References can be reassigned to point to some other object.

- `a` is just a name.
- `'abcd'` is an immutable object.
- `a = 'abcd'` creates a binding of the name `a` to the object `'abcd'`.
- `a = a + 'efgh'` creates a new object `'abcdefgh'` and binds `a` to it.

What does this mean?

Can we even do functional programming in Python?

Yes. We just need to be aware of the way Python works.

- Functional style, like object-oriented style, is a way of thinking.
- Techniques of writing code arise from each of these.
- Our goal is to add more ideas to our mental toolbelt, so that we can write code that implements higher degrees of complex behavior.

Monoids

Imagine a pair of rules that form a group.

```
add(a, b) = a + b where a and b are always integers
id = 0
```

With these rules, the following things are true:

```
add(10, 10) == 20          or       10 `add` 10 == 20
add(10, id) == 10          or       10 `add` `identity` == 10
```

Monoids

```
add(a, b) = a + b where a and b are always integers
id = 0
```

Notice that there's something else that is true about this group:

```
add(a, add(b, c) ) is the same as add( add(a, b), c)
```

This is called *associativity*. You can group monoid functions however you like, and the result will be the same.

Monoids

```
multiply(a, b) = a * b
id() = 1
```

```
8 `multiply` 8 == 64
8 `multiply` id() == 8
```

Why do we need the second of these two rules?

The simplest answer is that we need something that returns its input without modification. We need id() (or more properly, the *identity* or *unit* function) for the same reason we need: zero in addition, or one in multiplication, or an empty string in concatenation.

Monoids

- Monoids are the building blocks of functional programming.
- They are key to implementing features like recursion and iteration.
- They are excellent components for building pipelines.
- Monoids can be mixed and matched together to create complex behavior.
- The official term for this is *composition*.

Monads

```
m :: a -> Ma
```

Translation: m takes an a, and returns some transformation of an a.

The monad transformation can be computation, but it doesn't have to be. It can be anything, including an operation that has side effects.

Monads are the mechanism we use to contain actions that create or manipulate state.

Monads

Some of the most common uses of monads in a functional language:

- Handle Errors
- Print to the console
- Perform file operations
- Interact with a database
- Communicate over a network connection

Monads

The monad transformation can also add some *context* to the result. You can think of the transformed value as a value inside a container. The container *wraps* the value, and it also provides information above and beyond the value it contains.

Here's a metaphorical example:

- A slice of pizza (Item of data with no context)
- A slice of pizza in a microwave oven (Item of data with context)

Monads: Some Terms And Symbols

The monad *bind* operator: `>>=`

This is an operator that takes a value wrapped in a context, and passes it into a function. A few folks prefer to call it the *shove* operator.

`return`: This isn't the return statement we're familiar with. This return is an operation that wraps a value in a very basic, default context.

Monads: The Laws

In order for a system to be monadic, it needs to obey these rules:

- Left Identity: `return x >>= f` is the same as `f x`
- Right Identity: `m >>= return` is the same as `m`
- Associativity: `( m >>= f ) >>= g` is the same as `m >>= ( f >>= g )`

Monads: The Laws

What did that mean, in plain English?

- A value x, wrapped in a default context created by `return`, and then shoved into a function named f, is no different from calling f with the argument x.
- A monadic value m, shoved into `return`, results in our same monadic value m.
- A chain of monadic functions can be grouped or nested however you want, and the result will be the same.

Monad Example: Maybe

Let's look at how programming languages have historically dealt with error conditions.

- We check return values from function calls
- We use Try / Catch where it's available

Both of these methods leave a lot of work to be done by programmers.

Is this the best we can do?

Monad Example: Maybe

What if we had a way to represent a success or a failure that helped us make the whole system more fault tolerant? Let's imagine a function that follows a set of rules like this:

a -> Just a

a -> Nothing

- On success, it returns a result of type a, wrapped in a monadic context.
- On error, it returns a monadic context that signifies no result at all.

Monad Example: Maybe

If we compose a whole chain of functions that accept and return values *in context*, then we've taken a big step toward making our system more robust.

Imagine a space probe on its way to Mars. On its way past the Moon, the lunar gravity tugs at the probe and slightly alters its trajectory. The probe, however, is running a navigation routine powered by monads:

```
courseCorrection = invertVector(courseDeviation)
newCourse = currentCourse + courseCorrection
```

Monad Example: Maybe

CourseDeviation is a monad that contains an action: it reads an array of sensors and determines whether or not the craft is on course. If the craft is off course, the function will return a vector wrapped in a context, *Just V*. If the craft is on course, the function will return a context indicating no course deviation: *Nothing*.

How is this happening? Here's a bit of magic:

```
vectorOp v1 Nothing :: Nothing
vectorOp v1 v2 :: v -> v -> Just v
```

Monad Example: Maybe

```
vectorOp v1 Nothing :: Nothing
vectorOp v1 v2 :: v -> v -> Just v
```

What do these two rules mean? The first one means that any vector operation that receives a vector and a Nothing returns Nothing.

The second rule says that any vector operation that receives a vector and another vector will return a vector.

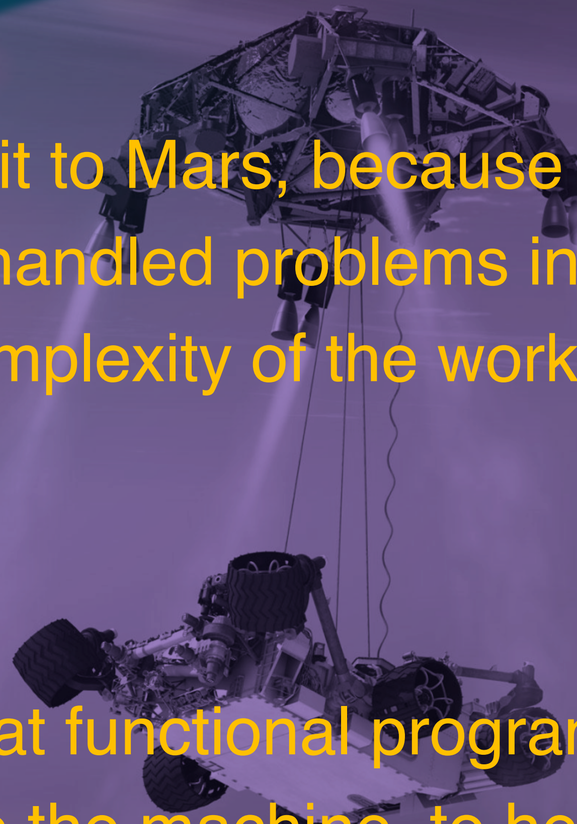
Monad Example: Maybe

```
vectorOp v1 Nothing :: Nothing
vectorOp v1 v2 :: v -> v -> Just v
```

This is really important: If we're on course, our course deviation is Nothing. When course correction runs, it takes Nothing, inverts it, and adds it to our current course. What is the result of inverting Nothing? *Nothing*. What's the result of adding Nothing to the vector that represents our current course? *Nothing*.

The result: if we're off course, we correct our trajectory. If not, we don't!

Conclusion



So our spacecraft makes it to Mars, because our code didn't break. And our code didn't break because we handled problems in a way that enlists the help of the machine to handle the complexity of the work.

This is the essence of what functional programming methods do for us: they allow human beings to leverage the machine, to help tell the machine what to do.

Thank You For Attending

FUNCTIONAL PROGRAMMING FOR THE BLUE TEAM

DEF CON 27

AUGUST 10, 2019